[T] [F] Modern operating systems are interrupt driven.

Enabling the interrupts is a privileged instruction.

[T] [F] I/O instructions and the instructions that turn interrupts on are generally considered to be privileged instructions.

The system call interface invokes

[T] [F] When using a text editor in UNIX, the characters that are typed are processed by the text editor application without any operating system activity other than processor scheduling.

[T] [F] The state of a process may transfer from “waiting” to “running” while an I/O operation completes.

[T] [F] The round-robin scheduling algorithm sometimes incurs starvation.

[T] [F] Kernel-scheduled threads are cheaper to create than user-level threads.

[T] [F] Different threads in the same process own an identical address space.

[T] [F] Shortest-Job-First scheduling algorithm is one of the usual algorithms used for CPU scheduling in modern operating systems.

[T] [F] With deferred cancellation scenario, a target thread can be cancelled safely.

[T] [F] The shortest-job-first algorithm is provably optimal because it gives the minimum average turnaround time for a given set of processes.

[T] [F] A solution to the critical section problem must satisfy the following requirements: Mutual exclusion, progress, and bounded waiting.

[T] [F] The operating system is not responsible for resource allocation between competing processes.

[T] [F] Contiguous memory allocation suffers from external fragmentation.

[T] [F] The state of a process may switch from waiting to running while an I/O completion.

(a) What are the differences between I/O bound and CPU bound processes? Describe one typical application for I/O bound processes and CPU bound processes, respectively.

(b) Draw the state transition diagram of a process.

(c) Describe the actions taken by a kernel to context-switch between processes.

(d) List and explain criteria we might use to evaluate a CPU scheduling algorithm.

(e) Describe the differences among long-term, medium-term and short-term process scheduling.

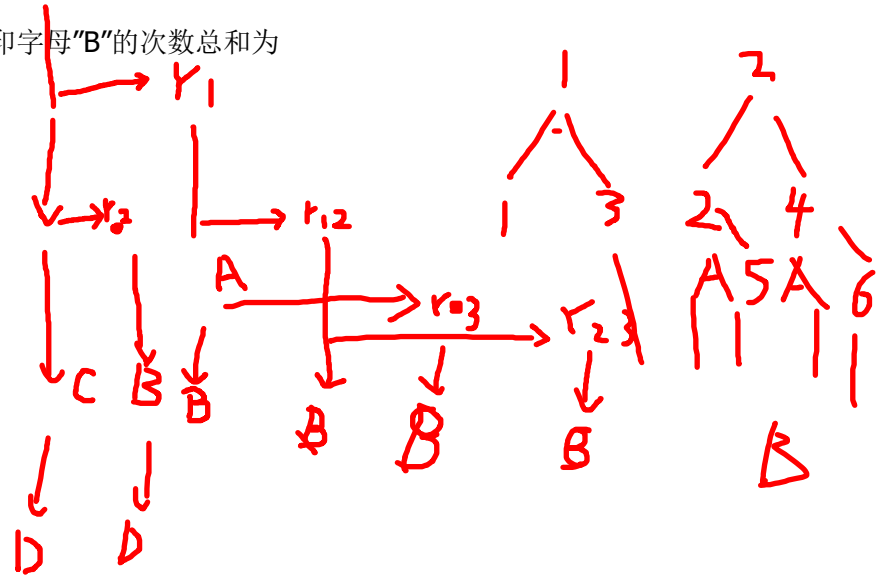
(f) What is convoy effect?

在 Shell 中执行如下程序，打印字母"B"的次数总和为

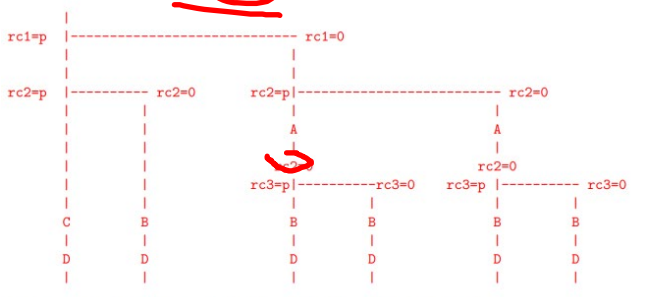
```
main()
{
   int rc1, rc2, rc3;
   rc1 = fork();
   rc2 = fork();

   if(rc1 == 0) {
      printf("A");
      rc2 = 0;
      rc3 = fork();
   }

   if(rc2 == 0){
      printf("B");
   }else{
      printf("C");
   }
   printf("D");
}
```



A、4　　B、5　　C、6　　D、7



Total number of printed　A's = 2,　B's = 5,　C's = 1 D's = 6

NOTE: If you compile and run these on a Linux/Unix machine you may need to insert fflush(stdout); after every printf otherwise you can get strange output (e.g., different numbers of letters because the output may be buffered and copied at the time of a fork.

1. **[20 points]** Five processes *A*, *B*, *C*, *D* and *E* arrived in this order at the same time with the following CPU burst and priority values. A smaller value means a higher priority.

	CPU Burst	*Priority*
A	3	3
B	7	5
C	5	1
D	2	4
E	6	2

Fill the entries of the following table with waiting time and average turn around time for each indicated scheduling policy. Ignore context switching overhead. Fill out the following table and provide a detailed elaboration. **Without a detailed elaboration, you risk to receive a very low or even 0 points.**

	Waiting Time					*Average*	
Scheduling Policy	*A*	*B*	*C*	*D*	*E*	*Waiting Time*	*Turn Around Time*
First-Come-First-Served	0	3	10	15	17	9	13
Non-Preemptive Shortest-Job First	2	16	5	0	10	6.8	11.2
Priority							
Round-Robin (time quantum=2)							

Answer: For each process, the following has the FCFS's start time, CPU burst (given), end time, waiting time and turnaround time:

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
A	0	3	3	0	3
B	3	7	10	3	10
C	10	5	15	10	15
D	15	2	17	15	17
E	17	6	23	17	23

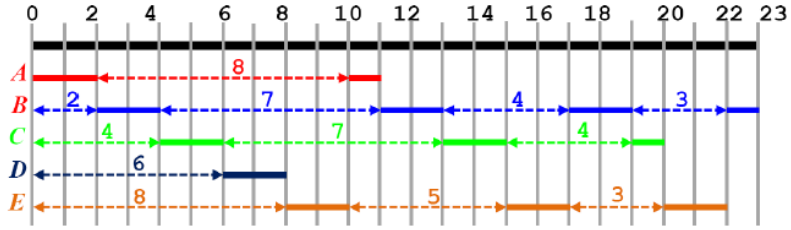
For each process, the following shows SJN's start time, CPU burst (given), end time, waiting time and turnaround time. Note that the order of processes is based on the length of CPU bursts, from the shortest CPU burst to the longest CPU burst.

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
D	0	2	2	0	2
A	2	3	5	2	5
C	5	5	10	5	10
E	10	6	16	10	16
B	16	7	23	16	23

For each process, the following is the Priority's start time, CPU burst (given), end time, waiting time and turnaround time. Note that the order of processes is based on the priority of each process, from highest to lowest.

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
C	0	5	5	0	5
E	5	6	11	5	11
A	11	3	14	11	14
D	14	2	16	14	16
B	16	7	23	16	23

The following figure shows the RR scheduling of these five processes. It is not difficult to see that A, B, C, D and E starts at time 0, 2, 4, 6 and 8, respectively, and has waiting times 8, $16 = 2+7+4+3$, $15 = 4+7+4$, 6, and $16 = 8+5+3$, respectively.



From this diagram, we are able to generate the following table:

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
A	0	3	11	8	11
B	2	7	23	16	23
C	4	5	20	15	20
D	6	2	8	6	8
E	8	6	22	16	22

The following table is a summary of our findings:

	Waiting Time					*Average*	
Scheduling Policy	*A*	*B*	*C*	*D*	*E*	*Waiting Time*	*Turn Around Time*
First-Come-First-Served	0	3	10	15	17	9.0=45/5	13.6 = 68/5
Non-Preemptive Shortest-Job First	2	16	5	0	10	6.6 = 33/5	11.2 = 56/5
Priority	11	16	0	14	5	9.2 = 46/5	13.8 = 69/5
Round-Robin (time quantum=2)	8	16	15	6	16	12.2 = 61/5	16.8 = 84/5

```
 1. int  intA;
 2. int  main(void)
 3. {
 4.   int    intB;
 5.   pid_t  cpid;

 6.   intA = 0;      intB = 2;
 7.   cpid = fork();
 8.   intA++;        intB++;
 9.   if (cpid == 0) {
10.      intA++;    intB++;
11.      printf("intA <%d>, intB <%d>\n", intA, intB);
12.   }
13.   else {
14.       wait();
15.       intA++;    intB++;
16.       printf("intA <%d>, intB <%d>\n", intA, intB);
17.    }
18. }
```

Suppose that all the calls execute normally.

- **[7 points]** Is the output from this program deterministic? More precisely, does it print out the same thing every time?
- **[8 points]** If the output is deterministic, what is printed out? (Why?) If not, explain as clear as possible what are the possible output?

Answer: Let us look at `intA` first. It is a global variable and initialized to `0` in the `main()` (line 6) before a child process is created. After a child process is created (line 7), the parent and the child have their own identical but separate address spaces. This means there are two copies of `intA`s, one in the parent's address space and the other in the child's. Then, both the parent and child see `intA` to be `0` in their own address spaces and add `1` to it. At this moment, `intA` is `1`. The child adds `1` to `intA` making its value `2` (line 10) and prints out the value of `intA`. Meanwhile, the parent may or may not be waiting for the child's completion. Once the child terminates, the parent adds `1` to its own copy of `intA`, making its value `2` (line 15). Therefore, for `intA` its value is always `2`.

Let us turn to `intB` which is set to `2` (line 6). A child process is created, and the parent and the child both have `intB` in their corresponding address spaces, and the value of `intB` is `2`. Finally, the parent (line 15) and the child (line 10) both add `1` to `intB`, and both copies of `intB` are the same (*i.e.*, `4`).

Therefore, the output of this program is **deterministic** and the out is

```
intA <2>, intB <4>
intA <2>, intB <4>
```

Note that there are two output lines, one from the parent and one from the child. ∎

[15 points] Suppose a system in which there are two types of processes, type *A* processes and type *B* processes. All processes of type *A* execute the same code, and all processes of type *B* execute the same code. The code for each process type is shown below.

A Processes	*B* Processes
P(X)	P(Y)
V(Y)	P(Y)
	V(X)
	V(Y)

Here, X and Y are general semaphores. X is initialized to 2, and Y is initialized to 0. Suppose three processes of type *A* and two processes of type *B* are brought into execution simultaneously. Answer the following two questions:

- **[8 points]** Is it possible for processes to finish in the order of *AABAB*? If so, show an execution sequence that results in this order. If not, explain why as accurate as possible.
- **[7 points]** Is it possible for processes to finish in the order *AABBA*. If so, show an execution sequence that results in this order. If not, explain why as accurate as possible.

Answer: The first execution order *AABAB* is possible, but the second one *AABBA* is impossible.

- The following is an execution sequence showing that *AABAB* is possible:

A_1	A_2	B_1	A_3	B_2	*Semaphore X*	*Semaphore Y*
					2	0
P(X)					1	0
V(Y)					1	1
	P(X)				0	1
	V(Y)				0	2
		P(Y)			0	1
		P(Y)			0	0
		V(X)			1	0
		V(Y)			1	1
			P(X)		0	1
			V(Y)		0	2
				P(Y)	0	1
				P(Y)	0	0
				V(X)	1	0
				V(Y)	1	1

  This execution sequence shows clearly that processes A_1, A_2 and A_3 in group *A*, and processes B_1 and B_2 in group *B* can indeed produce the order *AABAB*.

- The sequence *AABBA* is impossible. From the above execution sequence, after *AA* semaphores *X* and *Y* have counters 0 and 2. If *B* follows, *X* and *Y* will be 1 and 1. Because *Y*'s counter is 1, the next *B* can only pass its first P(Y) and blocks by the second P(Y). Then, the second *B* will be blocked by its first P(Y), and, as a result, the order of *AABBA* is impossible.

有两个并发执行的进程 P_1 和 P_2，共享初值为 2 的变量 x。P_1 对 x 加 1，P_2 对 x 减 1。加 1 和减 1 操作的指令序列分别如下所示。

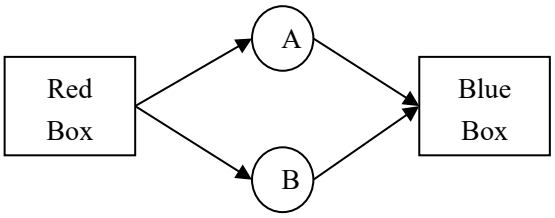
```
//加 1 操作                          //减 1 操作
load R1, x   //取 x 的值到寄存器 R1 中   load R2, x
inc R1                               dec R2
store x,R1   //将 R1 的内容存入 x      store x,R2
```

两个操作完成后，x 的值为（ C ）

A、可能为 0 或 4　　B、只能为 2

C、可能为 1、2 或 3　　D、可能为 0、1、2 或 3

Suppose there are two workers A and B in a factory. They take parts from the same red box and assemble them into machines. Then they put the assembled machines into another blue box. Make sure that they take parts from the red box mutual exclusively, and put machines into the blue box alternately (in the order: A, B, A, B …). Write C-like pseudo code that performs the appropriate initializations and enforces this execution order. Use only semaphores for your synchronization.




```
semaphore mutex, SA, SB;
mutex = 1; //用于互斥地从 red box 中取零件
SA = 1; //用于通知 A 可放入 blue box
SB = 1; //用于通知 B 可放入 blue box

A:
while (1)
{
    wait(mutex);
    从红筐中取零件；
    signal(mutex);
    组装机器；
    wait(SA);
    将组装的机器放入蓝筐；
    signal(SB);
}

B:
```

```
while (1)
{
    wait(mutex);
    从红筐中取零件；
    signal(mutex);
    组装机器；
    wait(SB);
    将组装的机器放入蓝筐；
    signal(SA);
}
```

Consider a multi-level feedback queue in a single-CPU system. The first level (queue 0) is given a quantum of 8 ms, the second one a quantum of 16 ms, the third is scheduled FCFS. Assume jobs arrive all at time zero with the following job times (in ms): 4, 7, 12, 20, 25 and 30, respectively. Show the Gantt chart for this system and compute the average waiting and turnaround time.